Assemblage automatique et adaptation d'applications à base de composants

Guillaume Grondin[*,**], Noury Bouraqadi[*] et Laurent Vercouter[**]

[*]Déptartement IA, École des Mines de Douai
941 rue Charles Bourseul – B.P. 10838, 59508 Douai Cedex, France
{grondin,bouraqadi}@ensm-douai.fr
http://csl.ensm-douai.fr
[**]Centre G2I, École des Mines de Saint-Étienne
158 cours Fauriel, 42023 Saint-Étienne Cedex 02, France
vercouter@emse.fr
http://www.emse.fr/~vercouter

Résumé. Dans cet article, nous introduisons MADCAR, un modèle de moteurs dédiés à la construction et à la reconfiguration dynamique et automatique d'applications à base de composants. Dans MADCAR, la description d'une application regroupe la définition des configurations valides et les règles de transfert de l'état de l'application lors des adaptations. Cette description est découplée de toute implémentation et peut donc être réutilisée avec différents jeux de composants. Partant d'une description d'application, un moteur MADCAR construit un problème de contraintes dont la résolution permet le choix de la configuration cible et des composants à utiliser. Ce choix prend en compte le coût de la configuration cible et son adéquation avec les ressources disponibles. Afin d'assurer la cohérence de l'application, le moteur utilise les règles de transfert d'état pour initialiser les attributs des composants de l'assemblage cible à partir des attributs des composants de l'assemblage de départ.

1 Introduction

L'adaptation est le processus par lequel un logiciel est modifié afin de prendre en compte un changement (Ketfi et al., 2002), que ce soit au niveau de l'environnement ou du logiciel lui-même. Il s'agit d'un processus en trois temps. Il faut *détecter* les changements, *décider* de la réaction la plus appropriée à la situation détectée, et enfin *réaliser* les traitements décidés. L'adaptation est dite *dynamique* si ce processus est réalisé sans arrêter l'exécution du logiciel. Cette dynamicité s'avère nécessaire dans différents domaines d'application (médecine, finances, télécommunications, ...) où un arrêt peut être très coûteux financièrement ou dangereux du point de vue humain ou environnemental. Elle est également requise pour concevoir des application autonomes capables de s'auto-adapter.

Nous nous intéressons dans ce papier aux étapes de décision et de réalisation des adaptations dynamiques dans les applications à base de composants (Szyperski, 2002). Dans ce

contexte, la réalisation d'une adaptation se traduit par une *reconfiguration*[1] de l'assemblage de composants constituant l'application. Cette reconfiguration consiste à ajouter, supprimer ou remplacer des composants ou des connexions entre composants. La reconfiguration de l'application englobe également la modification des attributs des composants et a fortiori le transfert d'état entre composants lors de ces reconfigurations.

Dans cet article, nous introduisons MADCAR[2] un modèle de moteurs dédiés à l'adaptation dynamique et automatique d'applications à base de composants. Ce modèle fournit une solution uniforme, basée sur un solveur de contraintes, permettant non seulement la construction d'applications par assemblage automatique mais aussi la reconfiguration dynamique de ces applications. La spécification des adaptations est à la fois globale à l'application et découplée des composants. De ce fait, l'approche MADCAR présente l'avantage d'éviter les problèmes de cohérence des adaptations rencontrés dans les approches où chaque composant s'adapte indépendamment des autres (David et Ledoux, 2003; Plasil et al., 1998). De plus, les spécifications d'assemblage et d'adaptation peuvent être réutilisées avec des jeux de composants différents, et ce pour des composants fournissant des contrats sur les ressources matérielles (CPU, mémoire, énergie) qu'ils requièrent. Enfin, comme notre approche d'adaptation est indépendante d'un modèle de composants particulier, nous proposons un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière générique. Hormis l'existence de certaines interfaces extra-fonctionnelles (contrats sur les ressources matérielles, gestion de l'état et de l'activité d'un composant), aucune hypothèse n'est faite sur les composants ou sur le modèle de composants utilisé.

Le reste de cet article se décompose de la manière suivante. La section 2 décrit les différentes parties du modèle MADCAR. La section 3 traite plus particulièrement de la gestion de l'état d'une application pendant les réassemblages. La section 4 décrit différents travaux liés à l'adaptation d'application à base de composants. Enfin la section 5 fournit un résumé de notre approche et énonce quelques perspectives.

2 Le modèle MADCAR

MADCAR (Grondin et al., 2006a,b) est un modèle de moteurs permettant l'assemblage et l'adaptation automatiques d'applications à base de composants logiciels. Les adaptations visées vont d'une simple modification de la valeur d'un attribut de composant jusqu'à un remplacement complet d'un assemblage de composants. Le concept de moteur d'assemblage se base sur des définitions existantes du problème d'assemblage automatique (Sora et al., 2003; Inverardi et Tivoli, 2002) : *étant donné un ensemble de composants et une description d'application, un* moteur d'assemblage *permet d'assembler des composants disponibles pour construire une application qui satisfait la description fournie.*

2.1 Hypothèses de travail

Les hypothèses minimales que nous faisons sont : l'homogénéité des composants en terme de modèle et l'auto-documentation des composants par contrats. Les contrats documentant les

[1] Parfois, nous employons le terme de « (ré)assemblage » pour désigner à la fois la construction d'un assemblage et la reconfiguration de cet assemblage.

[2] « Model for Automatic and Dynamic Component Assembly Reconfiguration ».

composants concernent les attributs paramétrables, les interfaces fournies ou requises, et les ressources matérielles requises pour leur fonctionnement (par exemple, les consommations en CPU, mémoire et énergie). A priori, notre approche est généralisable à l'utilisation de n'importe quel contrat portant sur les besoins matériels mais nous avons choisi ceux là car notre principal cadre d'application est l'informatique ubiquiste. La prise en charge de telles propriétés représente une des évolutions[3] de notre modèle comparé à Grondin et al. (2006a). Elle permet d'illustrer comment on peut prendre en compte dans MADCAR des contraintes extra-fonctionnelles lors des adaptations. Cette approche est également applicable aux composants qui ne fournissent aucun contrat sur les besoins matériels, notamment lorsque le domaine de l'application ne les utilise pas. Mais, dans ce cas, les décisions d'adaptations ne portent que sur la spécification fonctionnelle de l'application.

Notre approche s'attache à adapter une application ou un fragment d'application situé sur un seul site d'exécution. L'adaptation d'applications distribuées sort du cadre de ce papier.

Par ailleurs, nous ne nous intéressons dans ce travail qu'à la composition horizontale de composants car elle est applicable à n'importe quel modèle de composants. Notre approche permet l'utilisation de composants composites, mais elle ne prend pas en charge la composition hiérarchique des composants composites. Ce point permet d'abstraire des spécificités de chaque modèle de composants (voir la sous-section 2.2) pour pouvoir traiter - *i. e.* assembler et réassembler - des composants à travers un processus uniforme.

Ces hypothèses permettent de spécifier des applications à base de composants sans connaître le modèle concret qui sera utilisé pour les composants et leurs contrats[4], à la différence des approches plus spécialisées comme Chang et Collet (2007). Les spécifications d'assemblage que nous proposons ont vocation à être réutilisées telles quelles pour des jeux de composants issus de différents modèles de composants.

2.2 Le moteur d'assemblage

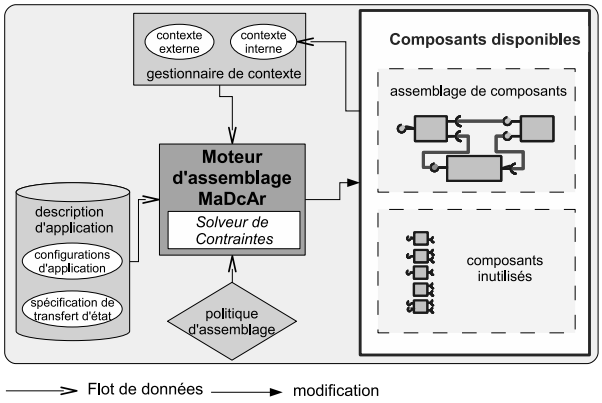

FIG. 1 – *Entrées-Sorties de* MADCAR.

[3]La deuxième évolution importante est le modèle de transfert d'état présenté dans la section 3.

[4]C'est ce découplage entre la spécification des applications et le modèle de composants utilisé qui explique la mise en valeur des lettres de MDA (« *Model Driven Architecture* », Bézivin et al. (2004)) dans l'acronyme MADCAR.

Comme illustré dans la figure 1, un **moteur d'assemblage** MADCAR permet de construire ou d'adapter une application à partir de quatre entrées : un *ensemble de composants* à assembler, une *description de l'application* qui représente la spécification des assemblages valides en termes de propriétés fonctionnelles et extra-fonctionnelles, une *politique d'assemblage* qui dirige les décisions d'adaptation et un *contexte* qui reflète aussi bien les informations sur les ressources matérielles (CPU disponibles, bande passante, etc.) que l'état de l'application (*i. e.* valeurs des attributs des composants assemblés).

Les composants, la description de l'application et le moteur d'assemblage peuvent être réutilisés pour construire des applications différentes. Nous suivons une *approche par « framework »* : un moteur d'assemblage MADCAR a une partie générique et une partie qui est spécifique au modèle de composants considéré. Pour prendre en charge un nouveau modèle de composants, il faut implanter un petit ensemble d'opérations typiques sur les composants : connexion/déconnexion de deux composants, activation/désactivation d'un composant et importation/exportation de l'état d'un composant.

La description d'application regroupe un ensemble de configurations alternatives et une spécification pour le transfert d'état. La description d'application et la politique d'assemblage sont spécifiées en termes de contraintes. Ainsi, un (ré)assemblage d'application avec MADCAR se traduit par un problème de satisfaction de contraintes (CSP[5], Kumar (1992)).

2.3 Le gestionnaire de contexte

Généralement, la définition d'un *contexte d'exécution* consiste à spécifier un ensemble de sondes logicielles qui peuvent fournir des « données pertinentes », c'est-à-dire des valeurs qui doivent être utilisées pendant un processus. En fait, « *Le Contexte n'est pas simplement l'état d'un environnement prédéfini avec un ensemble fixe de ressources d'interaction. Il fait partie d'un processus d'interaction avec un environnement en perpétuel changement composé de ressources reconfigurables, migrables, distribuées et multi-échelles* » (Coutaz et al., 2005). Par conséquent, la première entrée d'un moteur d'assemblage MADCAR est un **gestionnaire de contexte** qui modélise l'évolution d'un contexte d'exécution. Le gestionnaire de contexte doit définir un ensemble de sondes utilisées par un moteur d'assemblage non seulement pour affecter les décisions d'adaptation mais aussi pour déclencher les adaptations. De plus, il permet de définir la fréquence à laquelle les données contextuelles sont mises à jour et si nécessaire la taille de l'historique des valeurs mesurées par chaque sonde.

Les sondes du gestionnaire de contexte peuvent être en rapport avec les aspects internes ou externes de l'application. Le *contexte externe* d'une application inclut les informations sur les ressources matérielles (par exemple, CPU et mémoire), les réseaux disponibles (par exemple, bande passante courante et bande passante maximale) et des données géophysiques (par exemple, localisation et température). Le *contexte interne* d'une application est constitué des informations issues des composants de l'application : quels composants sont utilisés, quelles sont les connexions entre les composants et quel est l'état courant de l'application (valeurs des attributs de l'assemblage de composants). Pour les exemples utilisés dans ce papier, les sondes doivent fournir au minimum les niveaux courants pour le CPU disponible, la mémoire libre et l'énergie (pour une batterie) disponible sur l'infrastructure où sont déployés les

[5] « *Constraint Satisfaction Problem* ».

composants. Les valeurs de ces ressources matérielles seront respectivement notées $value_{cpu}$, $value_{memory}$ et $value_{energy}$.

À titre d'illustration, la figure 2 montre quelques spécifications de sondes. La sonde `networkAvailability` teste périodiquement si le réseau ethernet est disponible en évaluant une fonction `#isAvailable`. Cette sonde alimente le contexte externe de l'application. La sonde `bufferSize` récupère régulièrement la taille courante d'un attribut `buffer` depuis l'état de l'application (voir la sous-section 3.1). Dans cet exemple, chaque ressource observée possède sa propre période de mise à jour.

```
networkAvailabilitySensor := Sensor new.
networkAvailabilitySensor resource : 'external/hardware/network/ethernet'.
networkAvailabilitySensor operation : #isAvailable updatePeriod : 1000.
...
bufferSizeSensor := Sensor new.
bufferSizeSensor resource : 'internal/stateTransferNet/bufferNode'.
bufferSizeSensor operation : #getSize updatePeriod : 500.
```

FIG. 2 – *Un exemple de spécification de contexte.*

La conception d'un gestionnaire de contexte pose des problèmes typiques de performance, par exemple lorsqu'il y a plusieurs sondes qui doivent se mettre à jour très souvent. Ces questions sortent du cadre de ce papier. Dans la suite, nous supposons que les valeurs fournies par le gestionnaire de contexte utilisé sont toujours à jour lorsqu'elles sont lues par le moteur d'assemblage. Cette succincte description d'un gestionnaire de contexte est suffisante pour illustrer notre processus de réassemblage automatique. Pour plus de détails concernant la modélisation de contextes, on peut se référer à David et Ledoux (2005); Conan et al. (2007); Dey et al. (2001) qui décrivent des « *frameworks* » génériques de contexte.

2.4 Spécification des assemblages valides

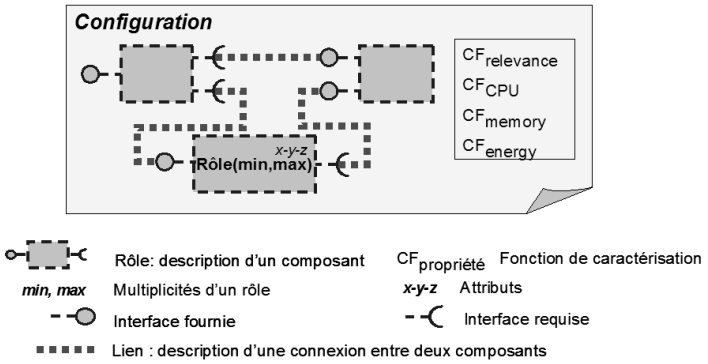

FIG. 3 – *Une configuration* MADCAR.

Une description d'application dans MADCAR définit l'ensemble des assemblages valides sous forme d'un ensemble de configurations alternatives. Chaque **configuration** MADCAR décrit une *famille d'assemblages* similaires car ayant les mêmes contraintes structurelles. Aucune

référence directe aux composants de l'application n'est introduite. En effet, chaque configuration se compose d'un graphe de rôles et d'un ensemble de fonctions de caractérisation, comme le montre la figure 3. Un **rôle** est une description de composant contenant un ensemble de *contrats* (Meyer, 1992; Beugnard et al., 1999). Ces contrats doivent au moins spécifier (a) un ensemble d'interfaces (requises ou fournies) qui symbolisent les interactions possibles avec d'autres rôles, (b) un ensemble d'attributs dont les valeurs permettent d'initialiser les composants et (c) deux multiplicités. Les multiplicités d'un rôle R définissent les nombres minimum (min) et maximum (max) de composants qui peuvent remplir simultanément ce rôle[6].

Un assemblage de composants se construit à partir d'une configuration. Chaque composant est sélectionné pour remplir un des rôles de cette configuration et les composants doivent être connectés entre eux selon les liaisons spécifiées entre les rôles qu'ils occupent. Dans MADCAR, les liaisons entre rôles ont une unique sémantique et ne se traduisent au niveau des composants que par une référence directe entre une interface fournie et une interface requise appartenant à deux composants distincts. Il est important de remarquer qu'un rôle est unique non seulement par ses interfaces et ses attributs, mais aussi par ses liaisons avec les autres rôles. Il en découle que *les composants qui remplissent un même rôle doivent être connectés exactement aux mêmes composants*. Cette règle permet de facilement automatiser la construction des assemblages à partir d'une configuration. Par exemple, l'assemblage résultant du lien entre $R_a(1,3)$ et $R_c(1,1)$ consiste à relier tous les composants jouant le rôle R_a au composant jouant le rôle R_c. Concernant les multiplicités des rôles, si le modèle de composants utilisé ne permet pas de connecter une interface fournie (resp. requise) à plusieurs interfaces requises (resp. fournies), alors les configurations dont certains rôles ont une multiplicité minimale strictement supérieure à 1 ne pourront pas être utilisées. Pour les autres configurations, il n'y aura qu'un composant par rôle.

Chaque configuration est caractérisée à travers des propriétés extra-fonctionnelles. La première propriété est la pertinence de la configuration pour un contexte donné (notée `relevance`). À celle-là, on peut ajouter diverses propriétés donnant par exemple le coût d'utilisation pour la configuration. Nous utilisons à titre d'exemple dans ce papier trois propriétés : le coût CPU d'une configuration (noté `CPU`), le coût mémoire d'une configuration (noté `memory`) et le coût en énergie d'une configuration (noté `energy`). Ces propriétés sont utilisées par le moteur d'assemblage afin de choisir une configuration lors d'un réassemblage. Pour chaque propriété P, le concepteur de configuration doit implanter une fonction de caractérisation CF_P qui mesure P. $CF_{relevance}$ mesure la pertinence d'une configuration par rapport à une situation contextuelle, c'est-à-dire à la fois les valeurs des sondes (contexte externe) et l'état de l'application (contexte interne). CF_{energy}, CF_{memory}, CF_{cpu} mesurent trois sortes de coûts pour la configuration étant donné un ensemble de composants, en supposant que les composants disponibles sont décrits par des contrats concernant les besoins en CPU, mémoire et énergie[7].

Pour que les propriétés de configuration soient comparables entre elles, un concepteur doit définir des fonctions de caractérisation normalisées. Les résultats doivent être compris entre 0 et 100. Des exemples de fonctions de caractérisation pour des propriétés de configuration sont donnés par les formules suivantes.

[6] Si besoin, nous noterons un rôle $R(min, max)$, où $min \geq 1$ et $max \geq min$.

[7] Si le domaine d'application ne requiert pas ce genre de préoccupations, alors les décisions ne se feront qu'en fonction de la pertinence et les composants n'auront pas à fournir d'informations sur leurs coûts.

$$CF_{relevance}(context) = \begin{cases} 100 & \text{, si } networkBandwidth \geq 56 \\ 50 & \text{, si } 0 < networkBandwidth < 56 \\ 0 & \text{, si } networkBandwidth = 0 \end{cases} \quad (1)$$

$$CF_{memory}(components, context) = \begin{cases} 0 & \text{, si } \frac{\sum_{c \in components} maxCost_{memory}(c)}{value_{memory}} \leq 1 \\ 100 & \text{, sinon} \end{cases} \quad (2)$$

Dans la formule 1, la pertinence d'une configuration dépend exclusivement de la bande passante d'un réseau. La formule 2 donne un exemple de fonction de caractérisation pour la propriété de coût « mémoire ». Elle se base sur le ratio entre (a) le coût mémoire maximum d'un assemblage basé sur la configuration étant donné un ensemble de composants, et (b) le niveau courant de mémoire libre ($value_{memory}$). La valeur de la quantité maximale de mémoire requise $maxCost_{memory}$ est supposée fournie par chaque composant.

2.5 La politique d'assemblage

Une politique d'assemblage permet de diriger les (ré)assemblages. Elle consiste à spécifier quand (*i. e.* dans quel contexte) et comment effectuer les assemblages. Dans MADCAR, la **politique d'assemblage** comporte deux parties utilisées respectivement pour (1) la détection des situations contextuelles qui peuvent nécessiter un (ré)assemblage, et (2) les règles qui guident les choix effectués par le moteur lors des réassemblages.

La première partie de la politique d'assemblage concerne les conditions requises pour *déclencher une adaptation*. Elle consiste dans la spécification de (1.a) une disjonction de plusieurs situations contextuelles et (1.b) la période minimale de temps entre deux évaluations des situations contextuelles données (notée `minimal-triggering-period`). Formellement, une situation contextuelle est une conjonction de contraintes portant sur des données contextuelles. Une adaptation est susceptible de survenir à chaque fois que le contexte courant correspond à une des situations contextuelles. Par exemple, le contexte $\{value_{CPU} = 2200; value_{memory} = 1024; value_{energy} = 60\}$ satisfait la situation contextuelle $(value_{CPU} > 1000)\&(value_{memory} \geq 512)$, mais ne satisfait pas la situation contextuelle $(value_{CPU} > 1500)\&(value_{energy} \leq 10)$. La valeur de `minimal-triggering-period` limite l'utilisation de ressources par le moteur d'assemblage et assure que chaque assemblage sera utilisé pendant une durée significative, comme le montre la figure 4. Il est de la responsabilité du concepteur de l'application de spécifier une valeur assez grande, en fonction de la taille de l'application et des ressources matérielles disponibles. Le choix d'une valeur trop petite conduirait à suspendre l'exécution de l'application pendant les adaptations.

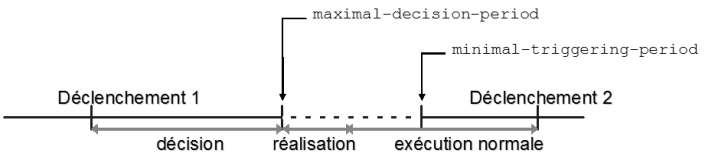

FIG. 4 – *Périodes d'adaptation et périodes d'exécution dans* MADCAR.

La seconde partie de la politique d'assemblage consiste en la spécification de (2.a) une fonction de sélection (notée SF_{bcc}) qui combine les fonctions de caractérisation d'une confi-

guration afin de déterminer leurs impacts respectifs sur les décisions d'adaptation de l'application, et (2.b) la période maximale de temps allouée au processus de décision (notée `maximal-decision-period`[8]). La sélection de la meilleure configuration et du meilleur ensemble de composants revient à maximiser la fonction de sélection SF_{bcc} fournie par le concepteur de l'application en fonction du contexte et des composants à choisir, jusqu'à ce que la durée `maximal-decision-period` ne s'achève. Un exemple de fonction de sélection est définie par la formule 3 qui est une simple fonction additive pondérée, mais le concepteur peut spécifier une fonction plus sophistiquée pourvu que les paramètres d'entrées soient les configurations, les composants et le contexte.

$$\begin{aligned} SF_{bcc}(components, context) = \quad & 40 * CF_{relevance}(context) \\ & -20 * CF_{energy}(components, context) \\ & -10 * CF_{memory}(components, context) \\ & -10 * CF_{cpu}(components, context) \end{aligned} \tag{3}$$

2.6 Le processus de (ré)assemblage

MADCAR peut être utilisé pour construire automatiquement des assemblages à partir de composants déconnectés et pour adapter (réassembler) dynamiquement et automatiquement des assemblages existants. Le processus de (ré)assemblage consiste en trois étapes successives : déclenchement, décision et réalisation.

1. *Déclenchement :* Le moteur d'assemblage vérifie périodiquement si le contexte courant correspond à une des situations contextuelles spécifiées dans la politique d'assemblage.
2. *Choix d'un assemblage pertinent :* Le moteur d'assemblage sélectionne la configuration et l'ensemble de composants qui maximisent la fonction de sélection SF_{bcc} spécifiée dans la politique d'assemblage. Cette maximisation est un problème de satisfaction de contraintes (CSP) avec :
   - une variable $V_{GoodConfig}$ dont le domaine est l'ensemble des configurations disponibles ;
   - un ensemble de variables V_{R_i} pour chaque rôle $R_i(min_i, max_i)$ des configurations avec pour domaine l'ensemble des composants disponibles et pour type une collection de taille comprise entre min_i et max_i ;
   - des contraintes sur chaque variable V_{R_i} qui sont les contrats du rôle R_i, auxquels on ajoute une contrainte qui assure que tous les composants sélectionnés pour les rôles doivent être distincts ;
   - une fonction objectif qu'il faut maximiser selon la politique d'assemblage : SF_{bcc}.

   Un unique CSP permet au moteur de sélectionner une des configurations et de sélectionner les composants à utiliser. Parmi les configurations qui sont éligibles[9] et pertinentes pour le contexte (c'est-à-dire où `relevance` est non nulle), le moteur doit choisir celle qui satisfait le mieux les préférences du concepteur.
3. *(Ré)assemblage :* Les composants actifs (associés à des processus légers) qui appartiennent à l'assemblage courant sont désactivés[10]. Ensuite, les nouveaux composants

[8] Il est nécessaire que `maximal-decision-period` < `minimal-triggering-period`.

[9] Une configuration est dite *éligible* lorsque chaque rôle de la configuration peut être rempli par un nombre de composants qui est au moins égal à la multiplicité minimale du rôle.

[10] Cela présuppose qu'ils soient dotés d'une interface extra-fonctionnelle permettant de contrôler leur activité. La désactivation/réactivation d'un ensemble de composants est reprise des travaux existants (Kramer et Magee, 1990).

sélectionnés sont assemblés et initialisés selon la configuration choisie. Finalement, les composants assemblés sont activés. Pendant cette étape, l'état de l'application est maintenu à travers un mécanisme de transfert d'état (voir la section 3).

3 Gestion de l'état d'une application lors des adaptations

La reconfiguration « à chaud » pose le classique, mais néanmoins épineux, problème de *transfert d'état* (Segal et Frieder, 1993). En effet, l'état de l'application doit rester cohérent quand il y a remplacement de composants. Nous décrivons ici un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière abstraite, indépendamment des composants logiciels. Un des problèmes auxquels nous devons faire face est que l'état ne peut être stocké dans les composants. Du fait du découplage entre les configurations et les composants, un même composant peut être utilisé pour remplir des rôles différents au fil des réassemblages.

3.1 Etat d'une application

L'état d'une application dans MADCAR s'appuie sur les attributs définis dans les rôles (voir la sous-section 2.4). Nous distinguons deux sortes d'attributs dans un rôle : les attributs fixes et les attributs variables. Un **attribut fixe** correspond à une donnée qui n'est utilisée que pour initialiser le composant qui va jouer ce rôle. Le composant ne modifie pas cette donnée. À titre d'exemple, la valeur maximale d'un compteur circulaire est un attribut fixe. Par contre, un **attribut variable** correspond à une donnée qui peut être modifiée par le composant. C'est le cas, par exemple, de la valeur courante d'un compteur.

Dans le modèle MADCAR, nous définissons l'**état d'une application** comme l'ensemble des valeurs de tous les attributs variables spécifiés des configurations. Les valeurs courantes de ces attributs correspondent à l'état courant de l'application. Lorsqu'un composant qui satisfait un rôle R est supprimé d'un assemblage, les attributs des composants qui sont marqués comme variables dans R doivent être sauvegardés. Ces données sont restaurées chez n'importe quel autre composant qui sera choisi pour remplir R par la suite. Notons que dans le cas des attributs d'un rôle ayant une multiplicité supérieure à un, c'est un vecteur de valeurs qui est sauvegardé pour mémoriser les attributs des composants jouant ce rôle. Un vecteur de taille n peut initialiser au plus n composants et s'il reste des composants non initialisés, alors on utilise une fonction d'initialisation définie dans chaque rôle.

3.2 Principe du transfert d'état

Dans une application, les attributs variables de deux configurations peuvent être liés. Il y a liaison entre deux attributs variables si la modification de la valeur de l'un doit être répercutée sur la valeur de l'autre pour que l'état de l'application reste cohérent. De telles liaisons relèvent de la sémantique de l'application et sont donc difficiles à déterminer de manière automatique. C'est donc au concepteur de l'application que revient la responsabilité de les spécifier. Pour ce faire, le transfert d'état doit être spécifié sous la forme d'un *réseau de transfert d'état*. Les nœuds de ce réseau sont les attributs variables. Les liens connectent uniquement des attributs définis dans des configurations différentes.

Le principal intérêt du réseau de transfert d'état est qu'il permet de déduire une séquence valide de transferts et de conversions pour la mise à jour cohérente des valeurs des attributs. C'est-à-dire une séquence qui préserve la cohérence de l'état de l'application et évite la perte d'information. En effet, certaines configurations utilisent des données qui ne sont pas utilisées dans d'autres configurations de l'application et des données qui sont présentes dans toutes les autres configurations, par exemple lorsqu'une configuration contient un rôle complètement différent des autres rôles des configurations. C'est pourquoi il faut maintenir les transferts d'états de manière globale dans un réseau.

D'après Vandewoude et Berbers (2003), on peut identifier deux grandes approches pour représenter l'état d'une application à base de composants : *l'approche basée sur un modèle de représentation* et *l'approche basée sur l'implémentation*. La première approche consiste à utiliser une entité « pivot » entre les composants qui sont interchangeables. Dans la deuxième approche, l'importation de l'état d'un composant doit être basée sur l'implémentation de l'export du composant à remplacer. Notre proposition pour MADCAR peut être vue comme une généralisation de la première approche. En effet, les rôles appartenant aux configurations jouent le rôle de pivots envers les composants, mais surtout le réseau de transfert d'état permet de spécifier les dépendances entre ces différents pivots. Par ailleurs, il faut noter que notre travail porte sur le transfert d'état de toute l'application dans le cas d'un changement potentiellement profond de l'architecture et ne se restreint pas au seul cas d'un remplacement de composant.

3.3 Spécification des règles de transfert d'état

Le transfert d'état doit être spécifié sous la forme d'un **réseau de transfert d'état**. Les nœuds de ce réseau sont les attributs variables. Les liens connectent uniquement des attributs définis dans des configurations différentes.

Chaque lien porte des *fonctions de transfert* pour calculer les valeurs d'un attribut à partir d'un autre. Étant données deux nœuds a et b reliés par un lien, le concepteur doit fournir deux fonctions de transfert : (1) $Transfer_{a-b}$ qui permet de calculer la valeur de b à partir de celle de a et (2) $Transfer_{b-a}$ qui permet de calculer la valeur de a à partir de celle de b.

Les liens sont orientés pour dénoter les risques de pertes d'informations. Considérons par exemple un lien orienté de a vers b (noté $a \triangleright b$). Cette orientation montre que a est *moins riche que* b, c'est-à-dire qu'il n'y a pas de perte d'information lorsqu'on transfère la valeur de b vers a et qu'en revanche il peut y avoir une perte d'information lorsqu'on transfère la valeur de a vers b. Pour éviter cette perte, la fonction de transfert $Transfer_{b-a}$ peut utiliser l'ancienne valeur de b pour calculer la nouvelle valeur.

Lorsqu'il y a un lien $a \triangleright b$, nous disons que b est maître de a et que a est esclave de b. Cette hiérarchie est transitive. Ainsi, si c est maître de b, alors c'est un maître indirect de a. Ainsi, la valeur de a peut être calculée à partir de c en passant par b et inversement. Nous parlons de *propagation* de valeurs. Un attribut maître qui n'a pas de maître est appelé un maître absolu. L'état d'une application peut être restreint à l'ensemble des maîtres absolus, puisque la valeur de tout autre d'attribut peut être déduite - par propagation - à partir des attributs maîtres absolus.

Pour illustration, considérons deux configurations dont la première contient un rôle avec un attribut variable url qui représente l'adresse internet d'un serveur et dont la seconde inclut un attribut variable $port$. Pour cet exemple, url est une chaîne de caractères composée de deux parties, un nom de serveur et le port du serveur, qui sont séparées par un caractère ' :'. Une

illustration du réseau de transfert d'état correspondant est montrée dans la figure 5. Les fonctions de transfert sont les suivantes :
$Transfer_{port-url}$: `url = url.cutBeforeLast(' :') + ' :' + port.toString`.
$Transfer_{url-port}$: `port = url.cutAfterLast(' :').toInteger`.

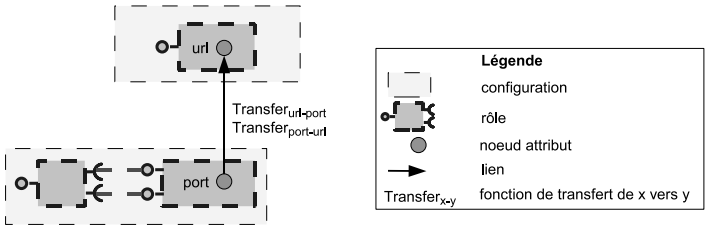

FIG. 5 – *Un simple réseau de transfert d'état dans* MADCAR.

Un réseau de transfert d'état peut comporter des liens multiples car une information dans une configuration peut être séparée en plusieurs dans d'autres configurations. Nous définissons deux liens multiples : un *lien join* qui connecte un groupe de nœuds vers un seul nœud, et un *lien fork* qui connecte un nœud vers un groupe de nœuds. De la même manière que pour les liens simples, les liens multiples portent des fonctions de transfert. Par exemple, un lien entre un nœud a et un groupe de nœuds $G_B = \{b_1, ..., b_j\}(j > 1)$ doit être « marqué » par deux fonctions de transfert : (1) $Transfer_{a-b_1,b_2,...,b_j}$ permettant de calculer les valeurs des nœuds du groupe G_B à partir de a et (2) $Transfer_{b_1,b_2,...,b_j-a}$ qui calcule a à partir de G_B.

Par ailleurs, les liens multiples sont également orientés. L'ensemble des nœuds cibles d'un lien *join* est appelé *groupe maître*, et l'ensemble des nœuds sources d'un lien *fork* est appelé *groupe esclave*.

3.4 Règles de cohérence

Afin de garantir la cohérence pendant les transferts d'état, les réseaux de transfert d'état doivent obéir aux règles suivantes :

- chaque attribut peut avoir au plus un maître direct : soit un attribut maître simple, soit un groupe d'attributs maîtres ;
- chaque attribut peut avoir au plus un esclave direct : soit un attribut esclave simple, soit un groupe d'attributs esclaves ;
- la connexion entre deux groupes de nœuds, passe obligatoirement par un *nœud virtuel*, c'est-à-dire un nœud qui ne correspond à aucun attribut dans les rôles des configurations ;
- les cycles sont interdits ;
- si des attributs $\{a_1, ..., a_j\}(j > 1)$ qui n'ont pas de maître sont liés et si aucun d'eux n'est moins riche que les autres, alors un nœud virtuel v doit être ajouté au réseau tel que $\{a_1, ..., a_j\} \triangleright v$ (*i. e.* lien *join* du groupe $\{a_1, ..., a_j\}$ vers un maître v).

Ces règles assurent qu'il est toujours possible de définir un ensemble d'attributs maîtres absolus. De plus, elles assurent que le chemin de transfert entre deux attributs - lorsqu'il y en a un - est unique puisque à partir de n'importe quel nœud, une seule fonction de transfert peut être utilisée pour propager une valeur vers d'autres nœuds (vers le haut ou vers le bas).

3.5 Transfert d'état

Le transfert d'état survient pendant un réassemblage. Il est réalisé en quatre phases :

1. lecture depuis les composants de l'assemblage courant des valeurs correspondant aux attributs variables pour chaque rôle de la configuration courante ;
2. propagation des valeurs des attributs variables de la configuration courante vers les attributs maîtres absolus de l'application en se basant sur le réseau de transfert d'état ;
3. propagation des valeurs des attributs variables des attributs maîtres absolus vers tous les attributs (notamment ceux de la nouvelle configuration) en se basant sur le réseau de transfert d'état ;
4. initialisation des attributs des composants qui ont été sélectionnés pour la nouvelle configuration : les valeurs des attributs variables proviennent du réseau de transfert d'état tandis que les valeurs des attributs fixes sont directement issues de leur spécification dans les rôles.

Le processus décrit ci-dessus s'opère quand il y a passage d'une configuration à une autre. Un cas particulier correspond au démarrage de l'application. Dans ce cas, aucune valeur n'est disponible pour les attributs fixes et variables de la configuration de démarrage. C'est pourquoi nous utilisons des fonctions d'initialisation. Chaque rôle contient une fonction d'initialisation pour ses attributs fixes et variables. De plus, les nœuds virtuels ont aussi leurs propres fonctions d'initialisation lorsqu'ils sont des maîtres absolus.

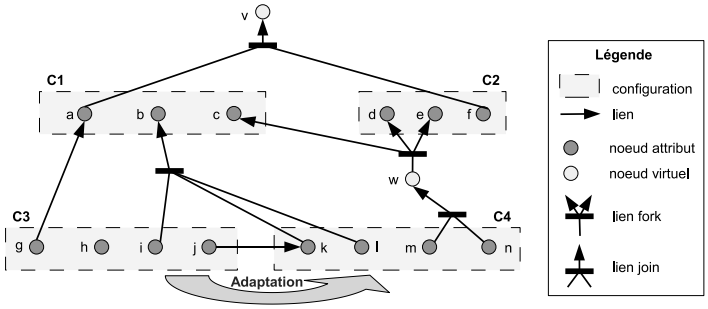

FIG. 6 – *Exemple de réseau de transfert. Les rôles et les fonctions de transfert ne sont pas indiqués.*

Un exemple de réseau de transfert d'état est montré dans la figure 6. Les attributs maîtres absolus sont v (qui est un nœud virtuel), b, c, d, e et h. Considérons l'initialisation de l'application sachant que la configuration choisie est $C3$. Elle se déroule en trois temps. Premièrement, les valeurs des maîtres absolus sont calculées à l'aide des fonctions d'initialisation puis propagées vers tous les autres nœuds du réseau de transfert d'état. Deuxièmement, il s'agit de compléter l'initialisation de la configuration $C3$. Pour chaque attribut variable des rôles de $C3$ qui ont une multiplicité maximale strictement supérieure à un, la fonction d'initialisation du rôle correspondant est utilisée pour que le vecteur de valeurs de cet attribut ait la même taille que le nombre de composants qui sont sélectionnés pour jouer ce rôle. Une fois l'initialisation de l'application passée, tous les attributs fixes et variables ont une valeur.

Considérons à présent une adaptation de la configuration $C3$ vers la configuration $C4$. La *première phase* du transfert d'état consiste à lire les valeurs de g, h, i et j depuis les composants qui remplissent les rôles de $C3$. Dans la *deuxième phase*, les valeurs des attributs de $C3$ sont propagées successivement jusqu'à un maître absolu et dans un ordre arbitraire (disons g, h, i, j). La valeur de g est propagée jusqu'à l'attribut a de $C1$ puis la valeur de a est propagée vers le maître absolu v grâce au lien *join* $\{a, f\} \triangleright v$ en utilisant la valeur temporaire de f et en appliquant $Transfer_{a,f-v}$. L'attribut h est un maître absolu et donc ne nécessite pas de propagation. La valeur de i est propagée vers b en appliquant $Transfer_{i,k,l-b}$, la valeur de j est propagée vers l'attribut k de $C4$ et la valeur de k est à son tour propagée vers b en appliquant $Transfer_{i,k,l-b}$ pour la deuxième fois[11]. La *troisième phase*, qui a lieu à la fin de la construction du nouvel assemblage, consiste à propager les valeurs des attributs maîtres absolus jusqu'aux attributs de l'ensemble des configurations. Par exemple, la valeur de b est propagée vers le bas pour calculer les valeurs de i, k et l. Dans la *quatrième phase*, chaque valeur des attributs de la configuration $C4$ est utilisée pour initialiser les composants du nouvel assemblage.

4 Travaux connexes

Plusieurs travaux existent concernant l'adaptation et l'auto-adaptation d'applications à base de composants. Nous présentons ici ceux qui nous semblent les plus aboutis dans ce domaine. On peut se référer à Senart (2003) pour avoir une vision plus complète des travaux portant sur l'adaptabilité des applications à base de composants.

SAFRAN (David et Ledoux, 2003) est une extension du modèle de composants Fractal[12] visant à supporter le développement de composants auto-adaptatifs. Développé à l'École des Mines de Nantes, SAFRAN est basé sur l'introduction d'une extension réflexive permettant de modifier de manière transparente le comportement d'un composant en fonction de son contexte d'exécution. L'adaptation de chaque composant est pilotée à l'aide d'une politique réactive individuelle basée sur des règles ECA (Évènement-Condition-Action). Cette approche peut conduire à des incohérences puisque chaque composant possède son propre ensemble de règles et s'adapte indépendamment des autres composants. De telles incohérences de l'application sont détectées a posteriori. Les adaptations en cours sont annulées et l'application est ramenée à son état précédant l'adaptation. Le coût de ces opérations imprévues peut être prohibitif, notamment dans le cas de systèmes contraints en ressources.

SOFA (« *SOFtware Appliance* ») est une plateforme développée à l'Université Charles en République Tchèque pour définir des applications distribuées à base de composants (Plasil et al., 1998). Dans SOFA, une application est vue comme une hiérarchie de composants. Les composants disposent d'une partie permanente qui est responsable de leur cycle de vie (notamment des mises à jour) et d'une partie remplaçable (code fonctionnel ou sous-composants). Mais, pour permettre ces remplacements (partiels) de composants, les développeurs doivent implémenter systématiquement certaines interfaces. Cette tâche est d'autant plus complexe quand il s'agit de réaliser des composants dynamiquement adaptables et de gérer les problèmes

[11] La première fois que la fonction $Transfer_{i,k,l-b}$ est appliquée, b est calculé à partir de la nouvelle valeur de i et des valeurs non modifiées de k et l. La deuxième fois, les nouvelles valeurs de i et k sont toutes les deux utilisées pour recalculer b. Comme l n'a pas été modifié, le calcul de b est terminé.

[12] `http://fractal.objectweb.org`

de cohérence qui en découlent. En outre, la nouvelle version d'un composant doit fournir les mêmes interfaces que l'ancienne version. Récemment, SOFA 2.0 (Bures et al., 2006) a été proposé pour pouvoir mieux contrôler les adaptations architecturales d'applications grâce à trois patrons de reconfiguration. Les patrons « *nested factory* », « *component removal* » et « *utility interface* » gèrent respectivement l'ajout de composants, la suppression de composants et les connexions entre composants quelles que soient leurs positions dans la hiérarchie de l'application. Enfin, la partie non remplaçable des composants est désormais reconfigurable avec une approche basée sur des micro-composants.

CASA (« *Contract-based Adaptive Software Architecture* ») est un « *framework* » développé par l'Université de Zurich permettant l'adaptation dynamique d'applications (Mukhija et Glinz, 2005). Pour réaliser l'adaptation des applications, CASA met en œuvre différents mécanismes. Chaque mécanisme est dédié à une cible d'adaptation particulière : les changements dynamiques dans les services de bas niveau (transmission de données et compression) utilisent des techniques réflexives, le tissage et le détissage dynamique d'aspects pour les adaptions transversales (comme changer le comportement de sécurité et de persistance) sont basés sur le système PROSE (Popovici et al., 2002), les changements dynamiques dans les attributs de l'application s'appuient sur des méthodes de « *callback* », et la recomposition dynamique de composants est dédiée aux adaptations qui impliquent l'ajout, la suppression et le remplacement de composants. Dans CASA, chaque adaptation peut impliquer plusieurs de ces mécanismes, selon la politique d'adaptation. Les auteurs de CASA travaillent avec leur propre modèle de composants et proposent un mécanisme original pour le remplacement dynamique de composants. Dans CASA, un composant est l'instance d'une classe. Et, un composant peut être remplacé dynamiquement par une autre instance de sa classe ou d'une classe alternative fixée à la conception. Une stratégie paresseuse de remplacement des composants permet de limiter fortement les problèmes d'incohérence. Une politique d'adaptation associe directement un contexte à une configuration appropriée. Les auteurs disent que la politique d'adaptation peut être modifiée dynamiquement car elle est dans un fichier XML séparé de l'application.

Contrairement à MADCAR, ces travaux sont spécifiques à un modèle de composants particulier. Dans ces travaux, le processus d'adaptation dynamique et automatique est entièrement contrôlé par le concepteur de l'application. De plus, la spécification des adaptations est exprimée à un haut niveau (grâce à une politique d'adaptation) dans le cas de MADCAR, SAFRAN et CASA. La spécification des adaptations dans MADCAR est à la fois globale à l'application (comme pour CASA) et découplée des composants. Dans SAFRAN et SOFA, elle est locale aux composants ou aux connecteurs, ce qui peut entraîner des problèmes d'incohérence ou dans le cas de SAFRAN un surcoût de performance dû à la correction a posteriori des adaptations incohérentes. MADCAR permet au concepteur de l'application d'avoir le contrôle sur la performance des adaptations car il fixe le temps alloué à la décision des adaptations. Concernant le transfert d'état lors des adaptations, MADCAR et CASA adoptent l'approche basée sur un modèle de représentation tandis que SAFRAN et SOFA utilisent l'approche (ad hoc) basée sur l'implémentation (voir la sous-section 3.2). Enfin, seul MADCAR offre l'avantage de pouvoir réutiliser les spécifications des adaptations définies par un concepteur avec des composants différents. L'utilisation d'un solveur de contraintes dans le processus de décision des adaptations marque la volonté de permettre au concepteur de spécifier les adaptations de manière déclarative mais sans avoir à raisonner au cas par cas sur les possibles situations contextuelles.

5 Conclusion et travaux futurs

Dans cet article, nous avons présenté MADCAR, un modèle de moteurs capable d'assembler et d'adapter automatiquement et dynamiquement des applications à base de composants. Dans MADCAR, un moteur d'assemblage possède quatre entrées : un ensemble de composants à assembler, une description de l'application qui représente la spécification de l'assemblage en termes de propriétés fonctionnelles et extra-fonctionnelles, une politique d'assemblage qui dirige les décisions d'adaptation et un contexte qui contient un ensemble de données (état de l'application, CPU disponible, bande passante, etc.) mesurées par des sondes. Au démarrage d'une application, le moteur MADCAR détermine une première configuration (description d'un assemblage) et un ensemble de composants à assembler pour construire l'application. Lorsque le contexte change, le moteur choisit une configuration plus appropriée et réassemble les composants disponibles en conséquence. Ainsi, le même mécanisme d'assemblage automatique s'applique à la fois à la construction des applications et à leur adaptation.

Une caractéristique majeure de MADCAR est qu'il permet aux concepteurs de produire des spécifications génériques. La spécification de l'architecture d'une application et de ses adaptations sont totalement découplées des composants logiciels. Aucune référence directe sur les composants n'est admise. Par ailleurs, la politique d'assemblage est séparée de la description de l'application. Ainsi, MADCAR encourage la séparation des préoccupations.

Par ailleurs, MADCAR prend en charge les adaptations non anticipées dans la mesure où les descriptions d'application et les composants peuvent être changés pendant l'exécution, c'est-à-dire sans stopper toute l'application. La spécification des adaptations est à la fois globale à l'application et découplée des composants. De ce fait, l'approche MADCAR présente l'avantage d'éviter les problèmes de cohérence des adaptations rencontrés dans les approches où chaque composant s'adapte indépendamment des autres. Enfin, les spécifications des adaptations peuvent être réutilisées avec des composants différents. Enfin, comme notre approche d'adaptation est indépendante d'un modèle de composants particulier, nous proposons un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière générique.

Nous avons développé un framework générique pour la mise en œuvre de MADCAR et nous l'avons appliqué au modèle de composants Fractal (Bruneton et al., 2002). L'outil résultant s'appelle *AutoFractal*[13]. Il est implémenté en Smalltalk et fonctionne sur les composants FracTalk[14] que nous développons dans l'équipe.

Dans nos travaux futurs, nous comptons porter nos efforts sur l'optimisation du moteur d'assemblage (coûts des décisions et des transferts d'état) pour les applications embarquées, afin d'adapter l'application malgré le faible niveau des ressources matérielles disponibles. Par ailleurs, nous envisageons d'étudier une évolution de MADCAR pour traiter spécifiquement le cas des applications distribuées.

Remerciements Nous tenons à remercier tous les relecteurs qui ont permis d'améliorer la présentation de ce travail.

[13] `http://csl.ensm-douai.fr/AutoFractal`

[14] Implantation Smalltalk du modèle de composants Fractal, `http://csl.ensm-douai.fr/FracTalk`

Références

Beugnard, A., J.-M. Jezequel, N. Plouzeau, et D. Watkins (1999). Making components contract aware. *Computer 32*(7), 38–45.

Bézivin, J., F. Jouault, P. Rosenthal, et P. Valduriez (2004). Modeling in the large and modeling in the small. In *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004*, pp. 33–46.

Bruneton, E., T. Coupaye, et J. Stefani (2002). Recursive and dynamic software composition with sharing. In *WCOP '02 : Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, Malaga, Spain.

Bures, T., P. Hnetynka, et F. Plasil (2006). Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *SERA '06 : Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications*, Washington, DC, USA, pp. 40–48. IEEE Computer Society.

Chang, H. et P. Collet (2007). Compositional Patterns of Non-Functional Properties for Contract Negotiation. *Journal of Software (JSW) 2*(2), 12.

Conan, D., R. Rouvoy, et L. Seinturier (2007). Scalable processing of context information with cosmos. In *DAIS '07 : Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems*, pp. 210–224.

Coutaz, J., J. L. Crowley, S. Dobson, et D. Garlan (2005). Context is key. *Commununication of the ACM 48*(3), 49–53.

David, P.-C. et T. Ledoux (2003). Towards a framework for self-adaptive component-based applications. In *DAIS '03 : Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems*, Volume 2893 of *LNCS*, pp. 1–14. Springer-Verlag.

David, P.-C. et T. Ledoux (2005). Wildcat : a generic framework for context-aware applications. In *MPAC '05 : Proceeding of th 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France.

Dey, A., D. Salber, et G. Abowd (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Special issue on context-aware computing in the Human-Computer Interaction Journal 16*(2–4), 97–166.

Grondin, G., N. Bouraqadi, et L. Vercouter (2006a). MADCAR : an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications. In *CBSE '06 : Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, Volume 4063 of *LNCS*, Vaesteras, Sweden, pp. 360–367. Springer-Verlag.

Grondin, G., N. Bouraqadi, et L. Vercouter (2006b). Assemblage Automatique de Composants pour la Construction d'Agents avec MADCAR. In *JMAC '06 : Journées Multi-Agent et Composant*, Nîmes, France, pp. 39–48. École des Mines d'Alès.

Inverardi, P. et M. Tivoli (2002). Correct and automatic assembly of COTS components : an architectural approach. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering (CBSE5) : Benchmarks for Predictable Assembly*.

Ketfi, A., N. Belkhatir, et P.-Y. Cunin (2002). Adapting applications on the fly. In *ASE '02 :*

Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, pp. 313. IEEE Computer Society.

Kramer, J. et J. Magee (1990). The evolving philosophers problem : Dynamic change management. *IEEE Transaction on Software Engineering 16*(11), 1293–1306.

Kumar, V. (1992). Algorithms for constraint satisfaction problems : A survey. *AI Magazine 13*(1), 32–44.

Meyer, B. (1992). Applying "design by contract". *Computer 25*(10), 40–51.

Mukhija, A. et M. Glinz (2005). Runtime adaptation of applications through dynamic recomposition of components. In *ARCS '05 : Proceedings of the 18th International Conference on Architecture of Computing Systems*, Innsbruck, Austria, pp. 124–138.

Plasil, F., D. Balek, et R. Janecek (1998). SOFA/DCUP : Architecture for component trading and dynamic update. In *ICCDS '98 : Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems*, pp. 35–42.

Popovici, A., T. Gross, et G. Alonso (2002). Dynamic weaving for aspect-oriented programming. In *AOSD '02 : Proceedings of the 1st international conference on Aspect-oriented software development*, New York, NY, USA, pp. 141–147. ACM Press.

Segal, M. E. et O. Frieder (1993). On-the-fly program modification : Systems for dynamic updating. *IEEE Software 10*(2), 53–65.

Senart, A. (2003). *Canevas logiciel pour la construction d'infrastructures logicielles dynamiquement adaptables*. Ph. D. thesis, Institut National Polytechnique de Grenoble.

Sora, I., F. Matthijs, Y. Berbers, et P. Verbaeten (2003). Automatic composition of systems from components with anonymous dependencies specified by semantic-unaware properties. *Technology of Object-Oriented Languages, Systems & Architectures 732*, 154–179.

Szyperski, C. (2002). *Component Software : Beyond Object-Oriented Programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Vandewoude, Y. et Y. Berbers (2003). Meta model driven state transfer in component oriented systems. In *Proceedings of The Second International Workshop On Unanticipated Software Evolution*, Warshau, Poland, pp. 3–8.

Summary

In this paper, we introduce MADCAR, a model of engines for dynamic and automatic (re)assembling of component-based software. In MADCAR, an application description consists of the definition of some valid configurations and the state transfer rules to apply during adaptations. This description is decoupled from any implementation and can therefore be reused with other components. Given an application description, a MADCAR engine builds a constraint solving problem that makes it possible to choose an appropriate configuration and the components to assemble. This choice takes into account the cost of the target configuration with respect to the available resources. To ensure the application consistency, the engine relies on the state transfer rules to initialize the component attributes of the target assembly using the component attributes of the source assembly.